Object Oriented Software Frameworks

Jim Kowalkowski

Framework Terms

Module: An object (in C++) that "plugs" into a processing stream and performs a specific task, without knowledge of how other modules work.

Framework: A object that creates, configures, and invokes modules in a sequence defined by the user at run time, without recompilation

What Does a Framework Do?

- Encapsulates a task within a module, such as a single reconstruction algorithm
- Enforces uniform configuration
- Centralizes common tasks
  - Reading, generating, and writing events
  - Managing database information
- Allows easy substitution of modules

Where Does it Apply?

- The event filter farm
- Simulation
- Offline reconstruction
- Trigger simulator
- First order analysis

Benefits of a Framework

- Developers do not need to know as many details about routine things
- Reduction of cut and paste coding and repetitive task coding
  - Reduced maintenance
  - Ability to upgrade in the future
- Compiler helps spot problems

Benefits Continued…

- Reduced development time
- Increased productivity
  - Configuration it similar
  - Running canned algorithm requires no code development
- Global and well defined handling of errors, timing, database, and memory

A Good Framework Will…

- Allow the development to focus on only the problem they are trying to solve
- Put low requirements on modules
- Make it easy to configure algorithms and jobs
- Enforce uniformity amongst applications
- Promote good use of C++
  - Exception handling
  - “construction is resource acquisition” idiom

A Good Framework Provides

- Classes that are specific to a single task
- Low coupling to services
- Ability to change dispatcher without affecting user code
- Low requirements to participate
- Flexibility and extendibility

Benefits of Consistency

- Applications work together
- Easier for developers to run all the various applications
- Mixing modules and applications works without using binding code or scripts

Inconsistency Consequences

- Producing disparate applications reduces productivity
  - Code for services repeated
- Interchange formats required
  - Time consuming to run
  - Time consuming to maintain
- Less overall code understanding
- Localized experts
- Adds rigidity to the system

Consequences of Limited Use

- Level 3 at DØ
  - Code needs to run in two frameworks
  - Two infrastructures to maintain
- GEANT4 independent of framework
  - Own concept of event, data objects
  - Translations required or carrying GEANT
  - Configuration dissimilarities
- Level 2 muon trigger simulator
  - Differing philosophies
  - Large amount of binding code

What Is Needed Early On?

- Clear concept of what a module is
- Configuration and algorithm parameter management
- Protocol for data exchange (EDM)
- Understanding of how conditions data will be treated and accessed (calibration, geometry, alignment, etc.)
- Requirements on sequencing

Early Goals for Design

- Defining module interfaces
- Error Logging and reporting
- Recovery and restart capabilities
  - Progress tracking
- Simple scheduling or sequencing
- Interactivity requirements
- Defining interfaces to database information

A Few Requirements

- Dynamic loading of modules (explicit and implicit)
- Sequences and subsequences which events flow through
- Intelligent propagation and handling of exceptions, messages, and program aborts
- Timing, memory leak checking
- Modules will be created with a sane state
- Module instances creates at runtime, as needed

More requirements

- No interactions between modules except through the event data
- Pick and choose active modules at run time
- Support multiple instances of modules
- Developer picks the functions of the module (reconstruction, filtering, analysis, display, run boundary processing)

Module Coding Guidelines

- No caching event data between events
- No global variables
- Do not use or contact other modules directly
- Do not cast away *const*-ness
- No circular dependencies
- No super modules that do everything

Problem Areas

- Event Display integration
- Scripting or interactive prompt
- Integration of simulation engine

Each competes for control of the main "event loop" or thread of control, each can put requirements on modules and data

Addressing these Areas

- Decide on the necessary requirements
- Consider multi-threading
  - Complex to produce infrastructure
  - Efficient solution to multiple event loops
- Consider multi-process
  - Interprocess communication is less efficient
  - Solves versioning and upgrade problems with libraries such as the event display

Some Observations with C++

- Configuring and running an executable that is already compiled is the easiest and safest thing to do
- Linking together a set of libraries to build a new executable is more difficult, requires time and build system knowledge
- Compiling a library and building an executable is the most error prone and time consuming process

Example Header (simplified)

```
Class TrackReco {
Public:
    TrackReco(const ConfigurationData& parms, Registry&);
    ~TrackReco();

    Directive reconstructEvent(Event& e);
    Directive analyzeEvent(const Event& e);
    Directive runStart(const RunInfo& I);

    Void Reconfigure(const ConfigurationData&);
Private: …
}
```

Example Implementation

```
TrackReco::TrackReco(const ConfigurationData& d,
    Registery& r) {
      r.subscribe("event reco",reconstructEvent);
      r.subscribe("event analysis",analyzeEvent);
      r.subscribe("run init",runStart);
      …
      double thresold = d.getDouble("thresold");
}

RegisterModule(TrackReco,"V1.0")
```

Notes:

- Division of labor is important – testing out algorithms or supporting multiple types for the same purpose is important
  - Want to reuse certain information in all the algorithms without reproducing or copying the code (hits or clusters in silicon or drift chambers).
  - Want to do the same analysis after the algorithms are run
- (arguments against super modules)